Solving the TTC'14 FIXML Case Study with SIGMA

Filip Křikava
University Lille 1 - LIFL
INRIA Lille, Nord Europe
France
filip.krikava@inria.fr

Philippe Collet
Université Nice - Sophia Antipolis
CNRS, I3S, UMR 7271
06900 Sophia Antipolis, France
philippe.collet@unice.fr

SIGMA is a family of Scala internal *Domain-Specific Languages* (DSLs) for model manipulation that provides expressive and efficient API for model consistency checking, model-to-model and model-to-text transformations. In this paper we describe a SIGMA solution for the *Transformation Tool Contest 2014* (TTC'14) FIXML case study, a transformation of FIXML XML format into class definitions in Java, C# and C++. The full case study including all three extensions have been realized and are publicly available on Github and in the SHARE environment.

1 Introduction

In this paper we describe the solution for the TTC'14 FIXML case study [9] using the SIGMA internal DSLs [8]. The case study involves *text-to-model* (T2M), *model-to-model* (M2M) and *model-to-text* (M2T) transformations, generating Java, C#, C++ code from a FIXML XML messages. Furthermore, we describe our approach to the three proposed extensions. The complete solution is available on a Github[1] as well as in the SHARE[2] environment in the virtual machine image `Ubuntu12LTS_TTC14_64bit_SIGMA.vdi`.

The solution is developed in SIGMA, a family of Scala[3] internal DSLs for model manipulation tasks such as model validation and model transformations. Developed as an open source project hosted on Github [4], SIGMA is a library that provides a dedicated Scala API allowing to manipulate models using high-level constructs similar to ones found in the external model manipulation DSLs such as ETL [7] or ATL [6]. The intent is to provide an approach that developers can use to implement many of the practical model manipulations within a familiar environment, reduced learning overhead and improved usability.

The solution uses the *Eclipse Modeling Framework* (EMF) [13], which is a popular meta-modeling framework widely used in both academia and industry. It is directly supported by SIGMA, however, other meta-modeling frameworks could be used as well, since SIGMA transformations are technologically agnostic.

[1] https://github.com/fikovnik/ttc14-fixml-sigma
[2] http://is.ieis.tue.nl/staff/pvgorp/share/
[3] http://scala-lang.org/
[4] https://fikovnik.github.io/Sigma

Submitted to:
TTC 2014

The remainder of this document is organized as follows: Section 2 gives a brief overview of SIGMA. Section 3 describes the solution for the case study core problem. Section 4 presents the solutions for the three case study extensions. Section 5 evaluates the solution using the evaluation criteria proposed in the case study, and finally Section 6 concludes the paper.

2 SIGMA Overview

SIGMA DSLs are embedded in Scala [11], a statically typed production-ready General-Purpose Language (GPL) that supports both object-oriented and functional styles of programming. Scala uses type inference to combine static type safety with a *"look and feel"* close to dynamically typed languages. It is interoperable with Java and has been designed to host internal DSLs [1]. Furthermore, it is supported by the major integrated development environments.

A typical way of embedding a shallow DSL into Scala is by designing a library that allows one to write fragments of code with domain-specific syntax. These fragments are woven within Scala own syntax so that it appears different [3]. Next to Scala flexible syntax (*e.g.* omitting semicolons and dots in method invocations, infix operator syntax for method calls, etc.), it has a number of features simplifying DSL embedding such as implicit type conversions allowing one to extend existing types with new methods, mixin-class composition (*i.e.* reusing a partial class definition in a new class) [11], and lifting static source information with implicit resolutions to customize error messages in terms of the domain-specific extensions using annotations [10].

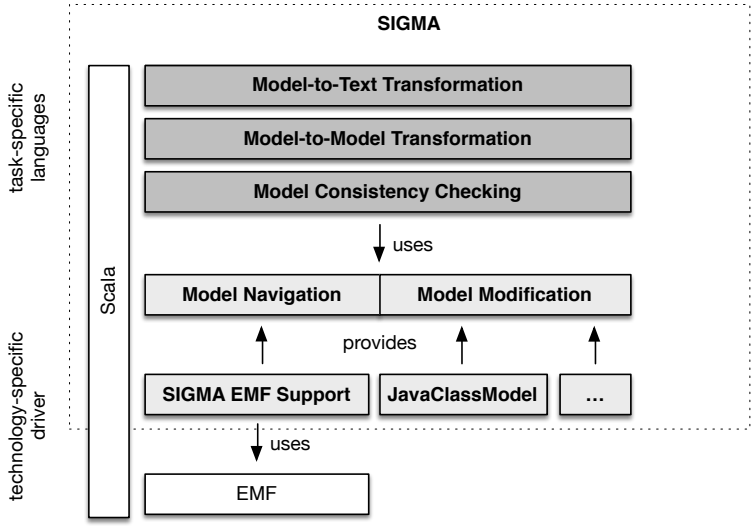

Figure 1: Sigma EMF to Scala Support

Figure 1 depicts the general organization of the SIGMA DSLs [8]. The use of EMF models in SIGMA is facilitated by a dedicated support layer that underneath uses the default EMF generated

Java classes and the EMF API. This layer provides a convenient model navigation and modification support forming a common infrastructure for the task-specific internal DSLs. While SIGMA targets the EMF platform, other meta-modeling platforms could be used as well. In Section 4.3 we will show how to create a support for a Java class model.

3 Solution Description

This section describes the solution for the core problem of transforming FIXML messages into Java, C# and C++ source code. As suggested, the solution is realized by systematic model transformations that are broken in the following tasks: (1) XML text to XML model (T2M transformation), (2) XML model to a model of an object oriented language (ObjLang) (M2M transformation), and (3) ObjLang model to source code (M2T transformation).

3.1 Overview

The input for the transformation chain is a file representing an FIXML message in the FIXML 4.4 version defined by the FIXML XML Schema [4]. The output is the corresponding Java, C# and C++ source codes that represent the data of the given FIXML message. The source concerning the solution to the core problem is located in the `ttc14-fixml-base` directory.

3.2 FIXML XML Message to XML Model (T2M)

The T2M transformation consists in parsing an FIXML XML document and creating an XML that conforms to the XML meta-model as specified in the case study description [9] (*cf.* Figure 2).

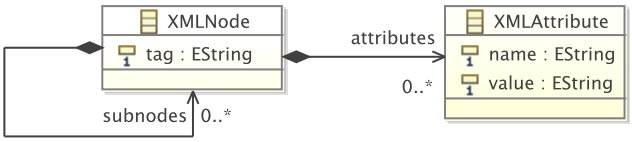

Figure 2: XML meta-model

Common Infrastructure. The first step before any SIGMA model manipulation task is to create the common infrastructure for the given model *i.e.* generate support classes that allows for seamless model navigation and modification using standard Scala expressions. In the case of EMF models, the common infrastructure aligns EMF generated Java classes with Scala. This involves model navigation without *"get noise"* (*e.g.* `node.getSubnodes` becomes `node.subnodes`), promoting EMF collections to corresponding Scala variants to benefit from convenient first-order logic operations (*e.g.*, map, filter, reduce), and first class constructs for creating and initializing new model elements.

This is addressed by generating extension traits that make EMF model elements interoperable with Scala. These traits implicitly extend all model classes with property accessors without the `get` prefix and convert EMF collections into the corresponding Scala ones. The conversion only happens at the interface level leaving the underlying data storage unchanged. In the same way, existing Scala types are extended with missing operations (*e.g.* `implies`). For example, the following is an excerpt[5] of the trait generated for the XML meta-model:

```
trait XMLMM extends EMFScalaSupport {
  implicit class XMLNode2Sigma(that: XMLNode) {
    def tag: String = that.getTag
    def tag_=(value: String): Unit = that.setTag(value)
    def subnodes: EList[XMLNode] = that.getSubnodes
    def attributes: EList[XMLAttribute] = that.getAttributes
  }
}
```

Furthermore, for each class in the meta-model, a Scala trait is generated that allows one to create the class instances in a concise way and also allows the classes to participate in Scala pattern matching constructs.

The `GenerateModelSupport` class is responsible for generating the common infrastructure for the EMF models used in this solution. It is a Scala executable object that first launches the standard EMF code generator to generate Ecore Java classes which is followed by the SIGMA common infrastructure generator. It is done in the way that the resulting sources go into the `src-gen` directory instead of the `src`, so that the generated code is separated from the user written one. This class has to be re-run every time any of the models changes.

Transformation. With the above model navigation and manipulation support, we can implement the actual transformation. Parsing XML is a common task and Scala already provides a solid support that is built into the language (*e.g.* XML literals). The object `FIXMLParser` is responsible for the transformation. It tries to parse FIXML message coming from various inputs (*e.g.*, text, file, input stream) and build the corresponding XML model. The actual transformation happens in the `parseNodes` and `parseAttributes` methods:

```
protected def parseNodes(nodes: Iterable[Node]): Iterable[XMLNode] = {
  val elems = nodes collect { case e: Elem => e }

  for (elem <- elems) yield XMLNode(
    tag = elem.label,
    subnodes = parseNodes(elem.child),
    attributes = parseAttributes(elem.attributes))
}

protected def parseAttributes(metaData: MetaData) =
  metaData collect {
    case e: xml.Attribute => XMLAttribute(name = e.key, value = e.value.toString)
  }
```

[5] XML meta-model generated code is in the `fr.inria.spirals.sigma.ttc14.fixml.xmlmm.support` package.

On the line 2 we discard any potential PCDATA nodes (*e.g.* white spaces) and then we simply yield a new `XMLNode` for `xml.Node` that has been parsed from XML. Only well-formed documents are considered since the underlying Scala XML library throws parsing exception in cases the input document is not well-formed. Additionally, we check for the presence of `<FIXML>` tag and provide a simple mechanism to discard FIXML messages that are not in the desired FIXML 4.4 Schema version.

3.3 XML Model to ObjLang Model (M2M)

This task involves transforming the XML model created in the previous section into a model of an object oriented language, ObjLang.

ObjLang meta-model. The meta-model of the ObjLang model used in this solution is shown in Figure 3. It originates from the Featherweight Java model [5], concretely from the version

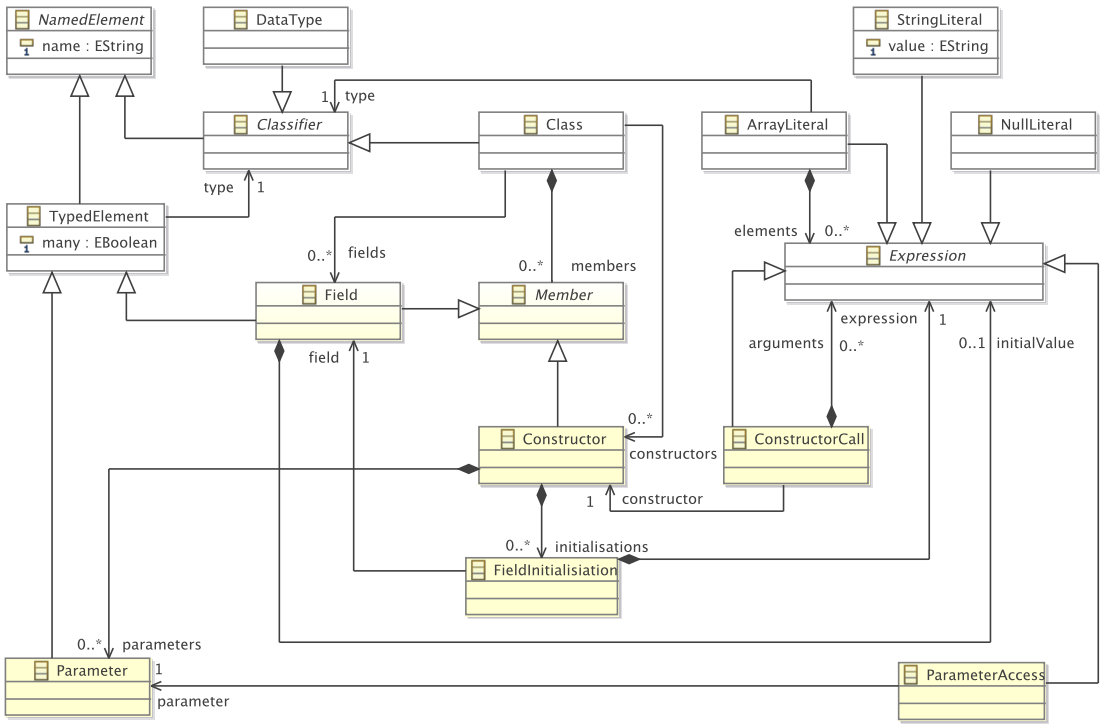

Figure 3: ObjLang meta-model

available at the EMFtext website[6]. It provides a reasonable abstraction over the close yet different models of the concerned programming languages. The model supports basic classes with fields, data types that are organized in a similar fashion as in Ecore (*i.e.* separating external types from model classes), and a basic set of expressions that is used for field initializations. The model itself more closely resembles a Java model than for example a C++ model. For example, a `Field` can optionally have an initial value, which is a feature not supported in C++[7]. These differences are left to be handled by the M2T transformers since the aim is to have just a single language agnostic meta-model.

Transformation. An M2M transformation provides necessary support for translating models into other models, essentially by mapping source model elements into corresponding target model elements. An imperative style of M2M transformation [2] is already supported thanks to the common infrastructure layer described above. On the other hand, the lower level of abstraction of the imperative transformation style leaves users to manually address issues such as orchestrating the transformation execution and resolving target elements against their source counterparts [7]. Therefore, inspired by ETL and ATL, we provide a dedicated internal DSL that combines the imperative features with declarative rule-based execution scheme into a hybrid M2M transformation language.

In SIGMA a M2M transformation is represented as a Scala class that inherits from the `M2MT` base class, which itself brings M2M DSL constructs into the class scope. Concretely, the `XMLMM2ObjLang` class is defined as:

```
1 class XMLMM2ObjLang extends M2MT with XMLMM with ObjLang {
2
3   sourceMetaModels = _xmlmm
4   targetMetaModels = _objlang
5
6   // transformation rules
7 }
```

Next to extending from the `M2MT` base class, it also mixes the `XMLMM` and `ObjLang` traits which are the generated support for the respective models participating in this transformation. In lines 3 and 4 it further specifies the transformation source and target models. In this case we translate one model into another one, but multiple models are supported.

The transformation rules are specified as methods. For example, the first rule of the transformation is defined as:

```
1 def ruleXMLNode2Class(s: XMLNode, t: Class) {
2   s.allSameSiblings foreach (associate(_, t))
3
4   t.name = s.tag
5   t.members ++= s.sTargets[Constructor]
6   t.members ++= s.allAttributes.sTarget[Field]
```

[6] http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_Featherweight_Java

[7] Initial values of fields in C++ are set in a constructor initializer list or in its body.

```
7   t.members ++= s.allSubnodes.sTarget[Field]
8 }
```

This method creates a rule named `XMLNode2Class` that transforms an `XMLNode` into a `Class`. A transformation rule in SIGMA may optionally define additional targets, but there is always one primary source to the target relation. This rule represents a matched rule which is automatically applied for all matching elements. When such a rule is executed, the transformation engine first creates all the defined target elements and then calls the method whose body populates their content using arbitrary Scala code. Inside the method body, additional target elements can be constructed (using the support provided by the common infrastructure), but in such a case, the developer is responsible for their proper containment and there will be no trace links associated with them.

A matched rule is applied once and only once for each matching source element, creating a 1:1 or 1:N mapping. However, this is not the case in the current scenario where XML document may contain multiple sibling elements with the same tag name which all should be mapped into an exact same class. This is done by associating all the same-tag siblings to the very same class during the rule application on the first of them (line 2). The method `allSameSiblings` is a helper method that collects all the elements that have the same tag and are at the same level.

The next four lines (4-7) populates the content of the class. The expression on the line 5 `t.members ++= s.sTargets[Constructor]` assigns all constructors that can be transformed from the source XML node into the class. Similarly `t.members ++= s.allAttributes.sTarget[Field]` add fields that have been transformed from the XML node attributes. The two methods `sTarget` and `sTargets` are defined on all model elements. They provide a way to relate the corresponding target elements that have been already or can be transformed from source elements. The difference between them is that `sTarget` should be used in the case of 1:N relationship between the source and target as opposed to 1:1 in the case of `sTarget`.

As for constructors, they are transformed using the following two rules:

```
1  def ruleXMLNode2DefaultConstructor(s: XMLNode, t: Constructor) {
2    s.allSameSiblings foreach (associate(_, t))
3  }
4
5  def ruleXMLNode2NonDefaultConstructor(s: XMLNode, t: Constructor) = guardedBy {
6    !s.isEmptyLeaf
7  } transform {
8
9    s.allSameSiblings foreach (associate(_, t))
10
11   for (e <- (s.allAttributes ++ s.allSubnodes.distinctBy(_.tag))) {
12     val param = e.sTarget[Parameter]
13     val field = e.sTarget[Field]
14
15     t.parameters += param
16     t.initialisations += FieldInitialisiation(
17       field = field,
18       expression = ParameterAccess(parameter = param))
19   }
20 }
```

The first one denotes a default (zero-argument) constructor. As we have discussed earlier, the ObjLang favors field initialization to constructor initialization and therefore the rule body is almost

empty. The single expression is the very same association that ensures that all same-tag sibling nodes maps to the same default constructor.

Unlike the default constructor, the rule creating a non-default constructor should only be applicable in the case there is at least one field to be set. In SIGMA this condition is represented by a rule guard that can further limit rule application using a boolean expression. The `!s.isEmptyLeaf` checks whether there is at least one attribute or a subnode in any of the same-tag siblings. It then creates a constructor parameter for each of the attributes and subnodes and uses them to initialize the class fields.

The constructor parameters are created using the following two rules, one for XML attribute and one for XML node. The `@LazyUnique` annotation denotes a lazy unique rule. Such rule is not automatically applied for matching elements and instead it has to be explicitly called using the `sTarget` or `sTargets` methods. Unlike lazy rule (annotated by `@Lazy`), it establishes a transformation trace between the source and targets and therefore it always returns the same targets for a given source.

```
1  @LazyUnique
2  def ruleXMLAttribute2ConstructorParameter(s: XMLAttribute, t: Parameter) {
3    t.name = checkName(s.name)
4    t.type_ = s.sTarget[Field].type_
5  }
6
7  @LazyUnique
8  def ruleXMLNode2ConstructorParameter(s: XMLNode, t: Parameter) {
9    val field = s.sTarget[Field]
10
11   t.name = field.name
12   t.many = field.many
13   t.type_ = field.type_
14 }
```

The class fields are populated from the `ruleXMLNode2Class` rule (lines 6 and 7). It calls the following two lazy unique rules:

```
1  @LazyUnique
2  def ruleXMLAttribute2Field(s: XMLAttribute, t: Field) {
3    t.name = checkName(s.name)
4
5    t.type_ = DTString
6    t.initialValue = StringLiteral(s.value)
7  }
8
9  @LazyUnique
10 def ruleXMLNode2Field(s: XMLNode, t: Field) {
11   val allSiblings = s.allSameSiblings
12   allSiblings foreach (associate(_, t))
13
14   t.type_ = s.sTarget[Class]
15
16   val groups = (s +: allSiblings) groupBy (_.eContainer)
17   val max = groups.values map (_.size) max
18
19   if (max > 1) {
20     t.name = s.tag + "_objects"
21     t.many = true
22     val init = ArrayLiteral(type_ = s.sTarget[Class])
23     val siblings = groups(s.eContainer)
```

```
24
25      init.elements ++= siblings.sTarget[ConstructorCall]
26      init.elements ++= 0 until (max - siblings.size) map (_ => NullLiteral())
27      t.initialValue = init
28    } else {
29      t.name = s.tag + "_object"
30      t.initialValue = s.sTarget[ConstructorCall]
31    }
32  }
```

The first one translates an XML attribute into a field. In the core problem specification, only string data types are used and therefore it assigns string data type regardless of the actual value of the attribute. In order not to conflict with language programming keywords, we provide a check that simply prepends the name with an underscore. The name conflicts could also be handled later at the M2T level, where each transformer can include a list of its language keywords. However, this could result in a state in which different names would be used for the same fields making the code inconsistent.

The second rule converts an XML node into a field. It has to handle the case of having a multiple same-tag siblings. The case description proposes to use either multiple fields initialized by specific constructors or by an array/list of such objects. While the former is easier to implement (simply by making the rule lazy), it creates a scalability problem since in Java, there is a limit of the maximum number of method parameters. For example the test case `test5.xml` already exceeds this number. Therefore we have opted for the latter solution and use arrays to represent multiple same-tag sibling nodes. The number of same-tag sibling nodes can vary within a parent node. For example:

```
<Pty ID="OCC" R="21"/>
<Pty ID="C" R="38">
        <Sub ID="ZZZ" Typ="2"/>
</Pty>
<Pty ID="C" R="38" Z="Q">
  <Sub ID="ZZZ" Typ="2"/>
  <Sub ID="ZZZ" Typ="3" Oed="X"/>
</Pty>
```

The `Sub` should be represented by an array field and the default initialization of `PosRpt` should equal to the following (in Java):

```
public Pty[] Pty_objects = new Pty[] {
  new Pty("OCC", "21", null, new Sub[] { null, null }),
  new Pty("C", "38", null, new Sub[] { new Sub("ZZZ", "2", null), null }),
  new Pty("C", "38", "Q", new Sub[] { new Sub("ZZZ", "2", null), new Sub("ZZZ", "3", "X") })
};
```

Note that the first and second instances of `Pty` contains two and one `null` respectively in the place of missing `Sub` subnode.

The `ConstructorCall` used for field initializations in the `ruleXMLNode2Field` is created from an XML node using the last rule in the transformation:

```
@Lazy
def ruleXMLNode2ConstructorCall(s: XMLNode, t: ConstructorCall) {
```

```
3   val constructor = s.sTargets[Constructor]
4     .find { c =>
5       (c.parameters.isEmpty && s.isEmptyLeaf) ||
6       (c.parameters.nonEmpty && !s.isEmptyLeaf)
7     }
8     .get
9
10  t.constructor = constructor
11
12  t.arguments ++= {
13    for {
14      param <- constructor.parameters
15      source = param.sSource.get
16    } yield {
17      source match {
18        case attr: XMLAttribute =>
19          // we can cast since attributes have always primitive types
20          val dataType = param.type_.asInstanceOf[DataType]
21
22          s.attributes
23            .find(_.name == attr.name)
24            .map { local => StringLiteral(local.value) }
25            .getOrElse(NullLiteral())
26
27        case node: XMLNode =>
28          s.subnodes.filter(_.tag == node.tag) match {
29
30            case Seq() if !param.many =>
31              NullLiteral()
32            case Seq(x) if !param.many =>
33              x.sTarget[ConstructorCall]
34            case Seq(xs @ _*) =>
35              val groups = (node +: node.allSameSiblings) groupBy (_.eContainer)
36              val max = groups.values map (_.size) max
37
38              val init = ArrayLiteral(type_ = param.type_)
39              init.elements ++= xs.sTarget[ConstructorCall]
40              init.elements ++= 0 until (max - xs.size) map (_ => NullLiteral())
41              init
42          }
43      }
44    }
45  }
46 }
```

First we need to find which constructor shall be used depending whether the given XML node (or any of its same-tag siblings) contains any attributes or subnodes. Next, we need to resolve the arguments for the case of non-default constructor. We do this by using the sources, *i.e.*, the source elements (XML node or XML attribute) that were used to create the constructor parameters. SIGMA provides `sSource` method that is the inverse of `sTarget` call with the difference that it will not trigger any rule execution. In the pattern matching we need to cover all possible cases such as an attribute defined locally or an attribute defined in a same-tag sibling, thus using `null` for its initialization.

3.4 ObjLang Model to Source code (M2T)

This task involves transforming the ObjLang model into source code for various programming languages. In the core problem solution, Java, C# and C++ are considered.

M2T transformations translate models into text by mapping source model elements into corre-

sponding textual fragments. SIGMA provides a template-based approach whereby string patterns are extended with executable logic for code selection and iterative expansion [2]. Unlike most of the external DSLs for M2T transformation, SIGMA uses the code-explicit form, *i.e.*, it is the output text instead of the transformation code that is escaped. From our experience, in non-trivial code generations, the quantity of text producing logic usually outweighs the text being produced. For the parts where there is more text than logic we rely on Scala multi-line string literals and string interpolations allowing one to embed variable references and expressions directly into strings.

Since we target multiple programming languages at the same time, we organize the code generation in a set of Scala classes and rely on Scala class inheritance and class mix-ins to compose, in a modular way, configuration for the respective languages. In the base class `BaseObjLangMTT` we define methods that synthesize expressions and data types. The default (Java) implementation can be then easily overwritten by other languages. Next we abstract a class generation into `BaseObjLang2Class`. The code generation is split into a fine-grained templates, represented as regular Scala methods, that generates fields, constructors and the like:

```
1  abstract class BaseObjLang2Class extends BaseObjLangMTT {
2
3    type Source = Class
4
5    def main = {
6      header
7
8      !s"class ${source.name}" curlyIndent {
9        genFields
10
11       !endl
12
13       genConstructors
14     }
15
16     footer
17   }
18
19   def genConstructors =
20     source.constructors foreach genConstructor
21
22   def genFields =
23     source.fields foreach genField
24
25   def genField(c: Field) =
26     !s"public ${type2Code(c)} ${c.name}${toInitCode(c)};"
27
28   def toInitCode(f: Field) =
29     " = " + (f.initialValue map toCode getOrElse (""))
30
31   def genConstructor(c: Constructor) = {
32     val args = c.parameters map param2Code mkString (", ")
33
34     !s"public ${source.name}($args)" curlyIndent {
35       c.initialisations foreach genFieldInitialization
36     }
37
38     !endl
39   }
40
41   def genFieldInitialization(fi: FieldInitialisiation) =
42     !s"this.${fi.field.name} = ${toCode(fi.expression)};"
43
44 }
```

Following the same pattern, a M2T transformation in SIGMA is a Scala class extending from the `M2T` base class. Line 3 defines the type of model element, *i.e.*, the transformation source. A M2T transformation consists in a set of templates that are represented as methods with the `main` method as the entry point. From there, a transformation is split and logically organized into smaller templates in order to increase modularity and readability. The most common operation in a M2T transformation is a text output. A convenient way to output text in our DSL is through a unary `!` (bang) operator that is provided on strings. The prefix `s` right before the string double quote denotes an interpolated string, which can include Scala expressions in a type-safe way.

Together with the **`ObjLang2Java`** class, a complete ObjLang to Java transformer can be instantiated. The class **`ObjLang2Java`** contains the Java language specifics which, in this case, is only the name of the `string` data type.

```
1 trait ObjLang2Java extends BaseObjLangMTT {
2   override def class2Code(p: DataType) = "String"
3 }
```

The generator for the C# is exactly the same and the class **`ObjLang2CSharp`** also only defines the `string` data type.

For C++, the situation is a bit more complicated, since we need to generate two files, a header and an implementation file. Also the C++ syntax differs from the one of Java a bit more. Furthermore, C++ does not support fields initialization and therefore the code generator must generate the correct initialization in the constructor. We use the C++11 standard since it supports inline array creation in the similar way as Java or C#, which greatly simplifies the code generation.

4 Extensions

In this section we describe our solutions to the case study extensions.

4.1 Extension 1 - Selection of Appropriate Data Types

The source concerning this is located in the `ttc14-fixml-extension-1` directory.

There are three changes needed in order to implement this extension.

(1) The first one is in the ObjLang meta-model where we need to add new expression classes representing literals for the new data types. In this extension we consider the following new data types: `double`, `long` and `integer`. While this list does not cover all the possible XML Schema data types, it provides a good basis for demonstrating how some support for additional ones could be added.

(1) The second one concerns the M2M transformation. We have to add the necessary support for guessing the data type of an attribute based on the string values of all of the same-tag siblings that have have the attribute in question. Following is the code snippet that realizes it:

```
            // basic types
            val DTString = DataType(name = "string")
            val DTDouble = DataType(name = "double")
            val DTLong = DataType(name = "long")
            val DTInteger = DataType(name = "int")

            // it also stores the promotion ordering from right to left
            val Builtins = Seq(DTString, DTDouble, DTLong, DTInteger)

            private val PDouble = """([+-]?\d+.\d+)""".r
            private val PInteger = """([+-]?\d+)""".r

            def guessDataType(value: String): DataType = value match {
              case PDouble(_) => DTDouble
              case PInteger(_) => Try(Integer.parseInt(value)) map (_ => DTInteger) getOrElse (DTLong)
              case _ => DTString
            }

            def guessDataType(values: Seq[String]): DataType =
              values map guessDataType reduce { (a, b) =>
                if (Builtins.indexOf(a) < Builtins.indexOf(a)) a else b
              }
```

(1) Finally, we need to update the code transformers to generate the appropriate data types. For example, in the C++ generator:

```
override def class2Code(p: DataType) = {
  import XMLMM2ObjLang._
  p match {
    case DTString => "std::string"
    case DTDouble => "double"
    case DTLong => "long"
    case DTInteger => "int"
  }
}
```

4.2 Extension 2 - Extension to Additional Languages

This extension adds a support for the C language. Since C is not an object oriented language, more work have to be done in the code generator part. It is important to node, that the M2M or T2M transformations have to remain untouched.

Instead of classes, in the case of the C language, we generate C structs with appropriate functions simulating object constructors. For each ObjLang class we generate a header file with a struct definition, a function for the struct creation and a set of functions representing class constructors. Following is an example of the C code synthesized from this FIXML message (for the `Pty` node):

```
<Pty ID="C" R="38">
  <Sub ID="ZZZZZ" Typ="2"/>
  <Sub ID="ZZZ" Typ="2"/>
</Pty>
```

```
#ifndef _Pty_H_
#define _Pty_H_

#include <stdlib.h>
```

```

#include "Sub.h"

typedef struct {
  char* _R;
  char* _ID;
  Sub** Sub_objects;
} Pty;

Pty* Pty_new();
Pty* Pty_init_custom(Pty* this, char* _R, char* _ID, Sub** Sub_objects);
Pty* Pty_init(Pty* this);

#endif // _Pty_H_
```

```
#include "arrays.h"

#include "Pty.h"

Pty* Pty_new() {
  return (Pty*) malloc(sizeof(Pty));
}

Pty* Pty_init_custom(Pty* this, char* _R, char* _ID, Sub** Sub_objects) {
  this->_R = _R;
  this->_ID = _ID;
  this->Sub_objects = Sub_objects;
  return this;
}

Pty* Pty_init(Pty* this) {
  this->_R = "21";
  this->_ID = "OCC";
  this->Sub_objects = (Sub**) new_array(2, Sub_init_custom(Sub_new(), "2", "ZZZ"), NULL);
  return this;
}
```

In order to simplify the code generator, we use a helper function `void **new_array(int size, ...)` that allows us to initialize arrays using simple expressions, which is not directly supported by C language or by the standard C library.

Despite the fact that C is not an object oriented language, our organization of the M2T transformation templates makes the implementation only 36 lines longer (30%) than the C++ version. The source concerning the extension 2 solution is in the `ttc14-fixml-extension-2` directory.

4.3 Extension 3 - Generic Transformation

The last extension aims at generic transformation between FIXML Schema and ObjLang model. Providing such a generic transformation essentially means creating a generic XML schema to ObjLang transformation. Such a task is far from being trivial and it would require a significant engineering effort. Therefore we have decided for an alternative solution in which we transform Java classes generated from an XML schema by the Java Architecture for XML Binding tool[8].

The advantage of this solution is that the JAXB already does all the hard work of parsing XML schema, resolving the element inheritance, substitutability, data types and others. The model of

[8] https://jaxb.java.net/

Java classes in an object oriented model and therefore the actual transformation into ObjLang is actually easier than in the case of the FIXML messages. Finally, the JAXB can be thought of as a another model transformation and therefore our solution is still within the model driven engineering domain.

The new input is a location of the FIXML XSD files and the new transformation workflow consists of the following stages:

(1) XSD to Java sources using JAXB.

(2) Compilation of Java source using regular Java compiler.

(3) Java model to ObjLang transformation.

(4) ObjLang to source transformation.

The implementation involves following changes:

(1) Extending the ObjLang model with a new classifier representing enumerated types.

(2) Extending the ObjLang model with the notion of a simple class inheritance.

(3) Extending the ObjLang model with the notion of abstract classes.

(4) A new M2M transformation between Java class model represented by the Java reflection API and ObjLang model.

(5) Extending the M2T transformation to cover the new ObjLang model concepts.

The new transformation is about 30% smaller than the one developed in the first extension and arguably less complex. It also demonstrates SIGMA support for manipulating different models than EMF. The source concerning the extension 3 solution is in the `ttc14-fixml-extension-3` directory.

One limitation of the current implementation is that we do not extend the C code generator to support the notion of class inheritance and for abstract classes. A potential solution is described in by Schreiner [12], however it requires to build a lot of supporting infrastructure which is out of the scope of this TTC'14 case. Similarly to the extension 1, we do not handle all the XML schema data types, but only the ones that appear in the FIXML schema.

5 Evaluation

In this section we evaluate the proposed solution using the evaluation criteria proposed in the case study description [9]. For the brevity, we only include the evaluation of the core problem solution. However, each extension is represented as a separate project and therefore further evaluation can be derived by comparing the changes required for their respective implementations.

Complexity. The case study description proposes to measure the complexity as the number of operator occurrences, features and entity type name references in the specification expressions.

To the best of our knowledge we do not know about any tool that can provide this metric for Scala code. We can therefore only provide our own estimate which is a rough number since it depends at which depth an expression is considered atomic. The M2M transformation of the core problem solution contains about 450 expressions. It uses 18 meta-models classes with 23 references.

Accuracy. The accuracy measures the syntactical correctness of the generated source code and how well does the code represents the FIXML messages.

The generated code is checked to be compiled for all valid examples provided in the case study. The compilation does not produce any warning and does not require any special compiler settings[9].

We used array representation of the same-tag sibling nodes which improves the quality of the code in comparison to numbered fields, and also provides better scalability. By implementing the first extension, we have further improved the usefulness of the generated code by guessing the appropriate type (*cf.* Section 4.1). While we do not cover the complete XML Schema type system, the solution is extensible and adding a support for a new data type is just a matter of providing the correct mapping into the corresponding languages. Finally, we have also implemented the generic FIXML Schema transformation that should result in a complete representation of FIXML messages in the different languages.

The case study did not require to provide additional features such as property getters and setters, or C/C++ destructors. As such the usability of the generated code is rather limited. However, the implementation of these features should be rather straightforward.

Development effort. Providing a correct measure of the complete development effort is not easy since by working on the case study we were also improving SIGMA at the same time. The solution for the code problem generating only Java code took about 4 person-hours in which about half was spent on designing the right ObjLang model by *trial-and-error*. Extending the code generator facilities for C# was almost for free as it involves only few lines of code. The C++ version was ready in less than one person-hour. The first extension was also finished in less than one hour since the ObjLang meta-model already supported multiple data types. The second extension involved more effort to correctly synthesize C code. Finally, the most work has been spent on the last extensions. In total, the development effort could be estimated to be about 15 person-hours.

Fault tolerance. A high fault tolerance is in cases that the transformation can detect invalid XML input and produce accurate results. In our solution, the well-formedness of the XML document is verified by the underlying Scala XML parser that provides a rather verbose error message indicating at which input location it found errors. Our parser provides additional checks that assert the

[9] The C/C++ `-fPIC` option is a standard option allowing the object files to be included in libraries, *cf.* http://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html#Code-Gen-Options

input document to be a FIXML 4.4 Schema version and not just any well-formed XML document or FIXML DTD version. This check is however rather simple, but could be easily extended by attaching a FIXML XML Schema location so the parser can verify the document correctness.

Execution time. The execution time for the transformations of the test data set provided by the case study is shown in Table 1. The times were collected by running the transformations in the SHARE virtualized environment `Ubuntu12LTS_TTC14_64bitSIGMA.vdi`. Therefore the following measures should be considered as micro-benchmark metrics with all the drawback that micro-benchmarking brings.

Message	T2M	M2M	M2T	Total
`test1`	9	70	191	270
`test2`	27	351	284	662
`test5`	1201	2734	459	4394
`test6`	376	1387	393	2156
Total	1613	4542	1327	7482

Table 1: Execution times for the test FIXML messages as measured on SHARE (in milliseconds)

Modularity. Modularity is defined as $1 - \frac{d}{r}$, where d is the number of dependencies between rules (implicit or explicit calls, ordering dependencies, inheritance or other forms of control or data dependence) and r is the number of rules [9].

The `XMLMM2ObjLang` for the core problem consists of $r = 8$ rules. There is one top level rule and all the other rules are calls, explicitly making $d = 7$. The modularity according to the formula is therefore 0.125.

6 Conclusion

In this paper we have presented a SIGMA solution for the TTC'14 FIXML case study. It covers the core problem of generating Java, C# and C++ code from FIXML messages including solutions for all of the case study extensions. The implementation is based on a systematic transformations of XML: (1) a text-to-model transformation of an XML input into an XML meta-model, (2) a model-to-model transformation of the XML meta-model into a object-oriented language-agnostic meta-model called ObjLang, and finally (3) a model-to-text transformation of ObjLang into a set of source code implementations for Java, C#, C++ and also C (in the second case study extension). We have specially opted for using a non-trivial ObjLang model in order to demonstrate a complex, yet expressive and quite concise transformation. The complete implementation of the core problem consists of 500 lines of Scala code[10]. Moreover, the ObjLang with the M2T generators

[10]The extension 1 consists of 550, extension 2 of 720 and extension 3 of 770 source lines of code

provides a generic object-oriented language model that could be likely reused in other model driven engineering scenarios.

In the third case study extension, we have implemented a generic FIXML Schema transformation into ObjLang model based on transforming annotated Java classes generated by Java Architecture for XML Binding.

Acknowledgment This work is partially supported by the Datalyse project www.datalyse.fr.

References

[1] Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky & Kunle Olukotun (2010): *Language virtualization for heterogeneous parallel computing*. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, ACM, doi:10.1145/1932682.1869527. Available at http://portal.acm.org/citation.cfm?doid=1932682.1869527.

[2] K. Czarnecki & S. Helsen (2006): *Feature-based survey of model transformation approaches*. *IBM Systems Journal* 45(3), pp. 621–645, doi:10.1147/sj.453.0621. Available at http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5386627.

[3] Gilles Dubochet (2011): *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. Ph.D. thesis, Ecole Polytechnique Fédérale de Lausanne.

[4] FIXML (2004): *FIXML 4.4 Schema Version Guide*.

[5] Atsushi Igarashi, Benjamin C Pierce & Philip Wadler (2001): *Featherweight Java: a minimal core calculus for Java and GJ*. *ACM Transactions on Programming Languages and Systems* 23(3), pp. 396–450, doi:10.1145/503502.503505. Available at http://portal.acm.org/citation.cfm?doid=503502.503505.

[6] Frédéric Jouault & Ivan Kurtev (2006): *Transforming Models with ATL*. In Jean-Michel Bruel, editor: *Satellite Events at the MoDELS 2005 Conference*, *Lecture Notes in Computer Science* 3844, Springer Berlin Heidelberg, pp. 128–138, doi:10.1007/11663430_14. Available at http://dx.doi.org/10.1007/11663430_14.

[7] D Kolovos, R Paige & F Polack (2008): *The Epsilon Transformation Language*. In: *Proceedings of the 2008 International Conference on Model Transformations*, Springer-Verlag, Zürich, Switzerland, pp. 46–60, doi:10.1007/978-3-540-69927-9_4. Available at http://www.springerlink.com/index/r575u280706p0371.pdf.

[8] Filip Krikava, Philippe Collet & Robert B France (2014): *Manipulating Models Using Internal Domain-Specific Languages*. In: *Symposium on Applied Computing (SAC), track on Programming Languages (PL)*, SAC.

[9] K. Lano, S. Yassipour-Tehrani & K. Maroukian (2014): *Case study: FIXML to Java, C\# and C++*. In: *Transformation Tool Contest 2014*.

[10] Adriaan Moors, Tiark Rompf, Philipp Haller & Martin Odersky (2012): *Scala-virtualized*. In: *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*,

PEPM '12, ACM, New York, NY, USA, pp. 117–120, doi:10.1145/2103746.2103769. Available at http://doi.acm.org/10.1145/2103746.2103769.

[11] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman & Matthias Zenger (2004): *An Overview of the Scala Programming Language*. Technical Report, École Polytechnique Fédérale de Lausanne.

[12] Axel-Tobias Schreiner (1993): *Object Oriented Programming with ANSI-C*. Axel-Tobias Schreiner. Available at http://www.cs.rit.edu/~ats/books/ooc.pdf.

[13] Dave Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks (2008): *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional. Available at http://www.eclipse.org/emf.